


C++ TUTORIAL - TEMPLATES - 2017



(<http://www.addthis.com/bookmark.php?v=250&username=khong7>)

K Hong
google.com/+KHongSanF
Francisc...



3,783 followers

Ph.D. / Golden Gate Ave, San Francisco / Seoul National Univ / Carnegie Mellon / UC Berkeley / DevOps / Deep Learning / Visualization

Sponsor Open Source development activities and free contents for everyone.



- K Hong (http://bogotobogo.com/about_us.php)

bogotobogo.com site search:

Templates

We use **templates** when we need functions/classes that apply the same algorithm to a several types. So we can use the same function/class regardless of the types of the argument or result.

In other words, a **template** is a mechanism that allows a programmer to use types as parameters for a class or a function. The compiler then generates a specific class or function when we later provide specific types as arguments. In a sense, templates provide **static (compile-time) polymorphism**, as opposed to dynamic (run-time) polymorphism. A function/class defined using template is called a **generic function/class**, and the ability to use and create generic functions/classes is one of the key features of C++.

We can think of a class template as a **type generator**. The process of generating types from a class template is called **specialization** or **template instantiation**.

In most of the cases, template instantiation is very complicated, but that complexity is in the domain of compiler writer, not the template user. Template instantiation takes place at compile time or link time, not at run time.

The syntax is:

```
template <class T> function_declaration;
```

or

```
template <typename T> function_declaration;
```

We can use either of construct. They are not different since both expressions have exactly the same meaning and behave exactly the same way.

Let's take one example of how a template generates a code. We have a function like this:

```
void f(vector<string>&v;)\n{\n    v.push_back("Steve Jobs");\n}
```

The compiler, when it sees **v.push_back("Steve Jobs")**, generates a function something like this ("Programming Principles and Practice Using C++" by Stroustrup, 2009)

Sponsor Open Source development activities and free contents for everyone.

Donate with **PayPal**

Thank you.

- K Hong (http://bogotobogo.com/about_us.php)

```
void vector<string>::push_back(const string& s){...}
```

from the following template definition:

```
template<typename T>
void vector<T>::push_back(const T& s){...}
```

The process of generating types from a given template arguments is called **specialization** or **template instantiation**. For example, `vector<int>` and `vector<Node*>` are said to be a specialization of `vector`.

Templates are the foundation of generic programming. Generic programming relies on polymorphism. Though there are several differences between OOP (class hierarchies and virtual functions) and generic programming (templates), the major difference is:

1. **Generic (templates)** : compile time resolution.
 - The choice of function invoked when we use is determined by the compiler at compile time.
2. **OOP (virtual functions)** : run time resolution.

Generic programming lets us write classes and functions that are polymorphic across unrelated types at compile time. A single class or function can be used to manipulate objects of a variety of types. The standard library containers, iterators, and algorithms are examples of generic programming. We can use library classes and functions on any kind of type.

When we parameterize a class, we get a **class template**, and when we parameterize a function, we get a **function template**.

So, what do people actually use template for?

1. When performance is essential.
2. When flexibility in combining information from several types is essential.

But note that, the flexibility and performance come at the cost of poor error diagnostics and poor error messages.

Terminology of Templates

Let's review the terminology of template using the following example implementing a generic stack class. This section of terminology is borrowed from "API design for C++" by Martin Reddy, 2011.

C++ Tutorials

C++ Home
(/cplusplus/cpptut.php)

Algorithms & Data Structures in C++ ...
(/Algorithms/algorithms.php)

Application (UI) - using Windows Forms (Visual Studio 2013/2012)
(/cplusplus/application_visual_stu

auto_ptr
(/cplusplus/autoptr.php)

Binary Tree Example Code
(/cplusplus/binarytree.php)

Blackjack with Qt
(/cplusplus/blackjackQT.php)

Boost - shared_ptr, weak_ptr, mpl, lambda, etc.

```
template <typename T>
class Stack
{
public:
    void push(T val);
    T pop();
    bool isEmpty() const;
private:
    std::vector<T> mStack;
};
```

1. Template Parameters

These names are listed after the template keyword in a template declarations. As shown in the example, **T** is the single template parameter specified in the **Stack** class.

2. Template Arguments

These are substituted for template parameters during specialization. In our example, given a specialization **Stack<int>**, the **int** is a template argument.

3. Instantiation

This is when the compiler generates a regular class, method, or function by substituting each of the template's parameters with a concrete type. This can happen implicitly when we create an object based on a template or explicitly if we want to control when the code generation happens. For instance, the following code creates two specific stack instances and will normally cause the compiler to generate code for these two different types:

```
Stack<T> myIntStack;
Stack<T> myStringStack;
```

4. Implicit Instantiation

This is when the compiler decides when to generate code for our template instances. Leaving the decision to the compiler means that it must find an appropriate place to insert the code, and it must also make sure that only one instance of the code exists to avoid duplicate symbol errors. This is non-trivial problem and can cause extra bloat in our object files or longer compile time. Most importantly, implicit instantiation means that we have to include the template definitions in our header files so that the compiler has an access to the definitions whenever it needs to generate the instantiation code.

5. Explicit Instantiation

This is when the programmer determine when the compiler should generate the code for a specific specialization. This can make for much more efficient compilation and link times because the compiler no longer needs to maintain bookkeeping information for all of its implicit instantiations. The onus, however, is then placed on the programmer to ensure that a particular specialization is explicitly instantiated only once. So, explicit instantiation allows us to move the template implementation into the **.cpp** file, and so hide from the user.

6. Lazy Instantiation

(/cplusplus/boost.php)

Boost.Asio (Socket Programming - Asynchronous TCP/IP)...
(/cplusplus/Boost/boost_Asynchl

Classes and Structs
(/cplusplus/class.php)

Constructor
(/cplusplus/constructor.php)

C++11(C++0x): rvalue references, move constructor, and lambda, etc.
(/cplusplus/cplusplus11.php)

C++ API Testing
(/cplusplus/cpptesting.php)

C++ Keywords - const, volatile, etc.
(/cplusplus/cplusplus_keywords.i

Debugging Crash & Memory Leak
(/cplusplus/CppCrashDebuggingI

Design Patterns in C++ ...
(/DesignPatterns/introduction.ph

Dynamic Cast Operator
(/cplusplus/dynamic_cast.php)

Eclipse CDT / JNI (Java Native Interface) / MinGW
(/cplusplus/eclipse_CDT_JNI_Min

Embedded Systems Programming I - Introduction
(/cplusplus/embeddedSystemsPr

Embedded Systems Programming II - gcc ARM Toolchain and Simple Code on Ubuntu and Fedora
(/cplusplus/embeddedSystemsPr

Embedded Systems Programming III - Eclipse CDT Plugin for gcc ARM Toolchain
(/cplusplus/embeddedSystemsPr

This describes the standard implicit instantiation behavior of a C++ compiler wherein it will only generate code for the parts of a template that are actually used. Given the previous two instantiations, for example, if we never called **isEmpty()** on the **myStringStack** object, then the compiler would not generate code for the **std::string** specialization of that method. This means that we can instantiate a template with a type that can be used by some, but not all methods of a class template. If one method uses the **>=** operator, but the type we want to instantiate does not define this operator. This is fine as long as we don't call the particular method that attempts to use the **>=** operator.

7. Specialization

When a template is instantiated, the resulting class, method, or function is called a specialization. More specifically, this is an instantiated (or generated) specialization. However, the term specialization can also be used when we provide a custom implementation for a function by specifying concrete types for all the template parameters. The following is called an explicit specialization:

```
template<>
void Stack<int>::push(int val)
{
    // integer specific push implementation
}
```

8. Partial Specialization

This is when we provide a specialization of the template for a subset of all possible cases. In other words, we specialize one feature of the template but still allow the user to specify other features. For example, if our template accepts multiple parameters, we could partially specialize it by defining a case where we specify a concrete type for only one of the parameters. In our **Stack** example with a single template parameter, we could partially specialize this template to specifically handle **pointers(*)** to any type **T**. This still lets users create a stack of any type, but it also lets us write specific logic to handle the case where users create a stack of pointers. This partially specialized class declaration looks like this:

```
template <typename T>
class Stack<T*>
{
public:
    void push(T* val);
    T* pop();
    bool isEmpty() const;
private:
    std::vector<T*> mStack;
};
```

Exceptions

(/cplusplus/exceptions.php)

Friend Functions and Friend Classes

(/cplusplus/friendclass.php)

fstream: input & output

(/cplusplus/fstream_input_output.php)

Function Overloading

(/cplusplus/function_overloading.php)

Functors (Function Objects) I - Introduction

(/cplusplus/functor_function_objects_i.php)

Functors (Function Objects) II - Converting function to functor

(/cplusplus/functor_function_objects_ii.php)

Functors (Function Objects) - General

(/cplusplus/functors.php)

Git and GitHub Express...

(/cplusplus/Git/Git_GitHub_Express.php)

GTest (Google Unit Test) with Visual Studio 2012

(/cplusplus/google_unit_test_gtest.php)

Inheritance & Virtual

Inheritance (multiple inheritance)

(/cplusplus/multipleinheritance.php)

Libraries - Static, Shared (Dynamic)

(/cplusplus/libraries.php)

Linked List Basics

(/cplusplus/linked_list_basics.php)

Linked List Examples

(/cplusplus/linkedlist.php)

make & CMake

(/cplusplus/make.php)

make (gnu)

(/cplusplus/gnumake.php)

Pros and Cons of Templates

1. Pros

1. It provides us **type-safe, efficient** generic containers and generic algorithms
2. The main reason for using C++ and templates is the trade-offs in performance and maintainability outweigh the bigger size of the resulting code and longer compile times.
3. The drawbacks of not using them are likely to be much greater.

2. Cons

1. Templates can lead to slower compile-times and possibly larger executable.
2. Compilers often produce incomprehensible poor error diagnostics and poor error messages.
3. The design of the STL collections tends to lead to a lot of copying of objects. The original smart pointer, `std::auto_ptr`, wasn't suitable for use in most collections. Things could be improved if we use other smart pointers from boost or C11.

Function Templates

Here is a simple example of how we use the template for a function which gives us the minimum value.

```
#include <iostream>

template <class T>
const T& min(const T& a, const T& b) {
    return a < b ? a : b;
}

int main()
{
    int x = 10, y = 20;
    long xLong = 100, yLong = 200;
    int minimum = min<int>(x, y);
    long minimumLong = min<long>(xLong, yLong);
    std::cout << "minimum = " << minimum << std::endl;
    std::cout << "minimumLong = " << minimumLong << std::endl;
}
```

If we drop `<int>` or `<long>`, the compiler still can do the job since it still can get the information about the type from the the function argument.

A function template can be declared **inline** in the same way as a nontemplate function. The specifier is placed following the template parameter list and before the return type. It shouldn't be placed in front of the **template** keyword. So, it looks like this:

```
template <class T>
inline const T& min(const T& a, const T& b) {
```

But **not** this:

Memory Allocation
(/cplusplus/memoryallocation.ph

Multi-Threaded Programming -
Terminology - Semaphore,
Mutex, Priority Inversion etc.
(/cplusplus/multithreaded.php)

Multi-Threaded Programming II
- Native Thread for Win32 (A)
(/cplusplus/multithreading_win3:

Multi-Threaded Programming II
- Native Thread for Win32 (B)
(/cplusplus/multithreading_win3:

Multi-Threaded Programming II
- Native Thread for Win32 (C)
(/cplusplus/multithreading_win3:

Multi-Threaded Programming II
- C++ Thread for Win32
(/cplusplus/multithreading_win3:

Multi-Threaded Programming
III - C/C++ Class Thread for
Pthreads
(/cplusplus/multithreading_pthre

MultiThreading/Parallel
Programming - IPC
(/cplusplus/multithreading_ipc.pl

Multi-Threaded Programming
with C++11 Part A (start, join(),
detach(), and ownership)
(/cplusplus/multithreaded4_cplu:

Multi-Threaded Programming
with C++11 Part B (Sharing
Data - mutex, and race
conditions, and deadlock)
(/cplusplus/multithreaded4_cplu:

Multithread Debugging
(/cplusplus/multithreadedDebug

Object Returning
(/cplusplus/object_returning.php

Object Slicing and Virtual Table
(/cplusplus/slicing.php)

```
inline template <class T>
const T& min(const T& a, const T& b) {
```

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets as in the example below.

```
template <class T, class U>
inline const T& min(const T& a, const U& b) {
    return a < b ? a : b;
}
```

Class Templates

Class can be generic using template.

```
template <class T>
class Complx {
private:
    T real, imag;
public:
    Complx(T&, T&);
    T& getReal();
    T& getImag();
};

template <class T>
Complx<T>::Complx(T& a, T& b) {
    real = a;
    imag = b;
}

template <class T>
T& Complx<T>::getReal() {
    return real;
}

template <class T>
T& Complx<T>::getImag() {
    return imag;
}

#include <iostream>

int main()
{
    double i=100, j=200;
    Complx <double> myComplx(i,j);

    std::cout <<"My complex is " << myComplx.getReal() << ", "
               << myComplx.getImag() << std::endl;
}
```

OpenCV with C++
(/cplusplus/opencv.php)

Operator Overloading I
(/cplusplus/operatoroverloading.

Operator Overloading II - self assignment
(/cplusplus/operator_oveloading.

Pass by Value vs. Pass by Reference
(/cplusplus/valuevsreference.php

Pointers
(/cplusplus/pointers.php)

Pointers II - void pointers & arrays
(/cplusplus/pointers2_voidpointe

Pointers III - pointer to function & multi-dimensional arrays
(/cplusplus/pointers3_function_n

Preprocessor - Macro
(/cplusplus/preprocessor_macro.

Private Inheritance
(/cplusplus/private_inheritance.p

Python & C++ with SIP
(/python/python_cpp_sip.php)

(Pseudo)-random numbers in C++
(/cplusplus/RandomNumbers.ph

References for Built-in Types
(/cplusplus/references.php)

Socket - Server & Client
(/cplusplus/sockets_server_client

Socket - Server & Client with Qt (Asynchronous / Multithreading / ThreadPool etc.)
(/cplusplus/sockets_server_client

Stack Unwinding
(/cplusplus/stackunwinding.php)

Standard Template Library (STL) I - Vector & List

Simple Template Exercise

Initial code

We have the following vector class:

(/cplusplus/stl_vector_list.php)

Standard Template Library
(STL) II - Maps
(/cplusplus/stl2_map.php)

Standard Template Library
(STL) II - unordered_map
(/cplusplus/stl2_unorderd_map_c

Standard Template Library
(STL) II - Sets
(/cplusplus/stl2B_set.php)

Standard Template Library
(STL) III - Iterators
(/cplusplus/stl3_iterators.php)

Standard Template Library
(STL) IV - Algorithms
(/cplusplus/stl4_algorithms.php)

Standard Template Library
(STL) V - Function Objects
(/cplusplus/stl5_function_objects

Static Variables and Static Class
Members
(/cplusplus/statics.php)

String (/cplusplus/string.php)

String II - sstream etc.
(/cplusplus/string2.php)

Taste of Assembly
(/cplusplus/assembly.php)

Templates
(/cplusplus/templates.php)

Template Specialization
(/cplusplus/template_specializati

Template Specialization - Traits
(/cplusplus/template_specializati

Template Implementation &
Compiler (.h or .cpp?)
(/cplusplus/template_declaration

The this Pointer
(/cplusplus/this_pointer.php)


```

class vector
{
    int sz;           // number of elements
    double* elem;     // address of first element
    int space;        // number of elements plus free-space
                      // slots for new elements

public:
    vector() : sz(0), elem(0), space(0) {}
    vector(int s) : sz(s), elem(new double[s]), space(s) {
        for(int i = 0; i < sz; ++i) elem[i] = 0;
    }
    vector(const vector&);
    vector& operator=(const vector & v);

    ~vector() { delete[] elem; }

    double& operator[](int n);
    const double& operator[](int n) const;

    int size() const { return sz; }
    int capacity() const { return space; }

    void reserve(int alloc_size);
    void resize(int resize_size);
    void push_back(double d);
};

void vector::reserve(int alloc_size) {
    if(alloc_size <= space) return;
    double *p = new double[alloc_size];
    for(int i = 0; i < sz; ++i) p[i] = elem[i];
    delete[] elem;
    elem = p;
    space = alloc_size;
}

vector& vector::operator=(const vector & v) {
    if(this == &v;) return *this;

    if(v.sz <= space) {
        for(int i = 0; i < v.sz; ++i)
            elem[i] = v.elem[i];
        sz = v.sz;
        return *this;
    }

    double *p = new double[v.sz];
    for(int i = 0; i < v.sz; ++i)
        p[i] = v.elem[i];
    delete[] elem;
    space = sz = v.sz;
    elem = p;
    return *this;
}

void vector::resize(int resize_size) {
    reserve(resize_size);
    for(int i = 0; i < resize_size; ++i) elem[i] = 0;
    sz = resize_size;
}

void vector::push_back(double d){
    if(space == 0)

```

Type Cast Operators
(/cplusplus/typecast.php)

Upcasting and Downcasting
(/cplusplus/upcasting_downcasti

Virtual Destructor &
boost::shared_ptr
(/cplusplus/virtual_destructors_sl

Virtual Functions
(/cplusplus/virtualfunctions.php)

*Programming Questions and
Solutions ↓*

Strings and Arrays
(/cplusplus/quiz_strings_arrays.p

Linked List
(/cplusplus/quiz_linkedlist.php)

Recursion
(/cplusplus/quiz_recursion.php)

Bit Manipulation
(/cplusplus/quiz_bit_manipulation

Small Programs (string,
memory functions etc.)
(/cplusplus/smallprograms.php)

Math & Probability
(/cplusplus/quiz_math_probabilit

Multithreading
(/cplusplus/quiz_multithreading.p

140 Questions by Google
(/cplusplus/google_interview_que

Qt 5 EXPRESS...
(/Qt/Qt5_Creating_QtQuick2_QM

Win32 DLL ...
(/Win32API/Win32API_DLL.php)

Articles On C++
(/cplusplus/cppNews.php)

```
        reserve(10);  
    else if(sz == space)  
        reserve(2*space);  
    elem[sz] = d;  
    ++sz;  
}
```

Let's convert the code using template parameters by replacing **double** with **T**:

What's new in C++11...
(/cplusplus/C11/C11_initializer_li

C++11 Threads EXPRESS...
(/cplusplus/C11/1_C11_creating_1

OpenCV...
(/OpenCV/opencv_3_tutorial_img

```

#include <iostream>

using namespace std;

template<class T>
class vector
{
    int sz;          // number of elements
    T* elem;         // address of first element
    int space;       // number of elements plus free-space

public:
    vector() : sz(0), elem(0), space(0) {}
    vector(int s) : sz(s), elem(new T[s]), space(s) {
        for(int i = 0; i < sz; ++i) elem[i] = 0;
    }
    vector(const vector&);
    vector& operator=(const vector & v);

    ~vector() { delete[] elem; }

    T& at(int n);
    const T& at(int n) const;

    T& operator[](int n);
    const T& operator[](int n) const;

    int size() const { return sz; }
    int capacity() const { return space; }

    void reserve(int alloc_size);
    void resize(int resize_size);
    void push_back(const T& d);
};

template<class T>
void vector<T>::reserve(int alloc_size) {
    if(alloc_size <= space) return;
    T *p = new T[alloc_size];
    for(int i = 0; i < sz; ++i) p[i] = elem[i];
    delete[] elem;
    elem = p;
    space = alloc_size;
}

template<class T>
T& vector<T>::at(int n) {
    if(n < 0 || sz <= n) throw out_of_range();
    return elem[n];
}

template<class T>
const T& vector<T>::at(int n) const {
    if(n < 0 || sz <= n) throw out_of_range();
    return elem[n];
}

template<class T>
T& vector<T>::operator[](int n) {
    return elem[n];
}

template<class T>

```

```

const T& vector<T>::operator[](int n) const {
    return elem[n];
}

template<class T>
vector<T>& vector<T>::operator=(const vector<T> & v) {
    if(this == &v;) return *this;

    if(v.sz <= space) {
        for(int i = 0; i < v.sz; ++i)
            elem[i] = v.elem[i];
        sz = v.sz;
        return *this;
    }

    T *p = new T[v.sz];
    for(int i = 0; i < v.sz; ++i)
        p[i] = v.elem[i];
    delete[] elem;
    space = sz = v.sz;
    elem = p;
    return *this;
}

template<class T>
void vector<T>::resize(int resize_size) {
    reserve(resize_size);
    for(int i = 0; i < resize_size; ++i) elem[i] = 0;
    sz = resize_size;
}

template<class T>
void vector<T>::push_back(const T& d){
    if(space == 0)
        reserve(10);
    else if(sz == space)
        reserve(2*space);
    elem[sz] = d;
    ++sz;
}

int main()
{
    vector<double> dv(3);
    dv.resize(5);
    for(int i = 0; i < dv.size(); ++i) {
        cout << dv[i] << " ";    //0 0 0 0 0
    }
    return 0;
}

```

One thing we should note in the **main()** is that we used the type **double** which has the default value of 0. For the **resize()** function, we could have done it by providing a way of initializing the vector:

```

template <class T>
void vector<T>::resize(int sz, T def=T());
...
vector<double> v
v.resize(10);    // add 10 copies of double() which is 0.0
v.resize(10, 2.0) // add 10 copies of 2.0

```

However, as shown in the example below, there is an issue of how do we handle a type when it does not have a default value. For instance, **vector<X>** where **X** does not have default value:

```
#include <vector>
struct X
{
    X(int) {};
    ~X() {};
};

int main()
{
    std::vector<X> xv;

    /* tries to make 3 Xs */
    //std::vector<X> xv2(3); // error: no appropriate default constructor available
    return 0;
}
```

So, in general, if we want to implement something like vector container where we do not have proper way of initialization for the type that does not have default value. Also, more trickier is the issue of destroying the elements when we are finished with them. So, let's look at what the standard library provides to handle those issues.

The allocator class

The standard library, fortunately, provides a class **allocator**, which provides **uninitialized memory**. The **allocator** class is a template that provides typed memory allocation, object construction and destruction. The **allocator** class separates **allocation** and **object construction**. When an **allocator** object allocates memory, it allocates space that is suitably sized and aligned to hold objects of the given type. However, the memory it allocates is unconstructed. Users of it should separately **construct** and **destroy** objects placed in the memory it allocates.

The class looks like this ("Programming Principles and Practice Using C++" by Stroustrup, 2009):

```
template<class T> class allocator
{
public:
    // allocate space for n objects of type T
    T* allocate(int n);

    // deallocate n objects of type T starting at p
    void deallocate(T* p, int n);

    // construct a T with the value v in p
    void construct(T* p, const T& v);

    // destroy the T in p
    void destroy(T* p);
};
```

The four operators allow us to:

1. **Allocate** memory of a size large enough to hold an object of type **T** without initializing.
2. **Deallocate** uninitialized space of a size large enough for an object of type **T**.
3. **Construct** an object of type **T** in uninitialized space. The **construct** function uses the copy constructor for type **T** to copy the value **v** into the element denoted by **p**.
4. **Destroy** an object of type **T**, thus returning its space to the uninitialized state.

The **allocator** is exactly what we need for implementing **vector<T>::reserve()**.

Converted code A using template

```

#include <memory>

template<class T>
class vector
{
    int sz;           // number of elements
    T* elem;          // address of first element
    int space;        // number of elements plus free-space
                     // slots for new elements

    std::allocator<T> a;

public:
    vector() : sz(0), elem(0), space(0) {}
    vector(int s) : sz(s), elem(new T[s]), space(s) {
        for(int i = 0; i < sz; ++i) elem[i] = 0;
    }
    vector(const vector&);
    vector& operator=(const vector & v);

    ~vector() { delete[] elem; }

    T& at(int n);
    const T& at(int n) const;

    T& operator[](int n);
    const T& operator[](int n) const;

    int size() const { return sz; }
    int capacity() const { return space; }

    void reserve(int alloc_size);
    void resize(int resize_size, T val);
    void push_back(const T& d);
};

template<class T>
void vector<T>::reserve(int alloc_size) {
    if(alloc_size <= space) return;
    T* p = a.allocate(alloc_size);
    for(int i = 0; i < sz; ++i) a.construct(&p[i], elem[i]);
    for(int i = 0; i < sz; ++i) a.destroy(&elem[i]);
    a.deallocate(elem, space);
    elem = p;
    space = alloc_size;
}

template<class T>
T& vector<T>::at(int n) {
    if(n < 0 || sz <= n) throw out_of_range();
    return elem[n];
}

template<class T>
const T& vector<T>::at(int n) const {
    if(n < 0 || sz <= n) throw out_of_range();
    return elem[n];
}

template<class T>
T& vector<T>::operator[](int n) {
    return elem[n];
}

```

```

template<class T>
const T& vector<T>::operator[](int n) const {
    return elem[n];
}

template<class T>
vector<T>& vector<T>::operator=(const vector<T> & v) {
    if(this == &v;) return *this;

    if(v.sz <= space) {
        for(int i = 0; i < v.sz; ++i)
            elem[i] = v.elem[i];
        sz = v.sz;
        return *this;
    }

    T* p = a.allocate(alloc_size);
    for(int i = 0; i < sz; ++i) a.constructa.destroy(&v.elem[i]);
    a.deallocate(elem, space);

    space = sz = v.sz;
    elem = p;
    return *this;
}

template<class T>
void vector<T>::resize(int resize_size, T val = T()) {
    reserve(resize_size);
    for(int i = 0; i < resize_size; ++i) a.construct(&elem[i], val);
    for(int i = 0; i < resize_size; ++i) a.destroy(&elem[i]);
    sz = resize_size;
}

template<class T>
void vector<T>::push_back(const T& d){
    if(space == 0)
        reserve(10);
    else if(sz == space)
        reserve(2*space);
    elem[sz] = d;
    ++sz;
}

```

Converted code B using template

We can start by passing **vector** an allocator parameter:

```

template<class T, class A = allocator<T>>
class vector
{
    A a;    // using allocate to handle memory for elements
    ...
};

```

With the handle defined as above line, the new code looks like this:


```

#include <memory>

template<class T, class A = allocator<T>>
class vector
{
    int sz;           // number of elements
    T* elem;          // address of first element
    int space;        // number of elements plus free-space
                     // slots for new elements

    A a;

public:
    vector() : sz(0), elem(0), space(0) {}
    vector(int s) : sz(s), elem(new T[s]), space(s) {
        for(int i = 0; i < sz; ++i) elem[i] = 0;
    }
    vector(const vector&);
    vector& operator=(const vector & v);

    ~vector() { delete[] elem; }

    T& at(int n);
    const T& at(int n) const;

    T& operator[](int n);
    const T& operator[](int n) const;

    int size() const { return sz; }
    int capacity() const { return space; }

    void reserve(int alloc_size);
    void resize(int resize_size, T val);
    void push_back(const T& d);
};

template<class T, class A >
void vector<T,A>::reserve(int alloc_size) {
    if(alloc_size <= space) return;
    T* p = a.allocate(alloc_size);
    for(int i = 0; i < sz; ++i) a.construct(&p[i], elem[i]);
    for(int i = 0; i < sz; ++i) a.destroy(&elem[i]);
    a.deallocate(elem,space);
    elem = p;
    space = alloc_size;
}

template<class T, class A>
T& vector<T,A>::at(int n) {
    if(n < 0 || sz <= n) throw out_of_range();
    return elem[n];
}

template<class T, class A>
const T& vector<T,A>::at(int n) const {
    if(n < 0 || sz <= n) throw out_of_range();
    return elem[n];
}

template<class T, class A>
T& vector<T,A>::operator[](int n) {
    return elem[n];
}

```

```

template<class T, class A>
const T& vector<T,A>::operator[](int n) const {
    return elem[n];
}

template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector<T,A> & v) {
    if(this == &v;) return *this;

    if(v.sz <= space) {
        for(int i = 0; i < v.sz; ++i)
            elem[i] = v.elem[i];
        sz = v.sz;
        return *this;
    }

    T* p = a.allocate(alloc_size);
    for(int i = 0; i < sz; ++i) a.construct(&p[i], v.elem[i]);
    for(int i = 0; i < sz; ++i) a.destroy(&v.elem[i]);
    a.deallocate(elem,space);

    space = sz = v.sz;
    elem = p;
    return *this;
}

template<class T, class A>
void vector<T,A>::resize(int resize_size, T val = T()) {
    reserve(resize_size);
    for(int i = 0; i < resize_size; ++i) a.construct(&elem[i], val);
    for(int i = 0; i < resize_size; ++i) a.destroy(&elem[i]);
    sz = resize_size;
}

template<class T, class A>
void vector<T,A>::push_back(const T& d){
    if(space == 0)
        reserve(10);
    else if(sz == space)
        reserve(2*space);
    elem[sz] = d;
    ++sz;
}

```

Nontype Template Parameters

It is useful to parameterize classes with types. How about parameterizing classes with **nontype** things, such as string values, integral expressions, pointers, and references?

We can use a parameter which is not a type. A nontype parameter is considered as part of the type. For the standard class **bitset**<>, we can pass the number of bits as the template argument.

```

bitset<32> b32;           // bitset with 32 bits
bitset<64> b64;           // bitset with 64 bits

```

The bitsets have different types because they use different template arguments. So, we can't assign or compare them.

This nontype parameter is replaced by value when the function is called. The type of that value is specified in the template parameter list. In the example below, the function template declares **arrayInit** as a function template with one type and one nontype template parameter. The function itself takes a single parameter, which is a reference to an array.

```
template <class T, size_t N>
void arrayInit(T (&a;)[N])
{
    for (size_t i = 0; i != N; ++i) a[i] = 0;
}

int main()
{
    int i[10];
    double d[20];
    arrayInit(i);    // arrayInit(int (&)[10])
    arrayInit(d);    // arrayInit(double (&)[20])
}
```

A template nontype parameter is a constant value inside the template definition. We can use a nontype parameter when we need constant expressions. In the example, we need that to specify the size of an array. So, when **arrayInit** is called, the compiler figures out the value of the nontype parameter from the array argument.

Here is another example using integer as a **non-type** parameters. In the code below, we want to assign values for all the elements of a vector. So, we set the template parameter with an integer value as non-type, and as we traverse each element of the vector container, we set the value we want by calling **setValue()**.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <int val>
void setValue(int& elem)
{
    elem = val;
}

class PrintElements
{
public:
    void operator()(int& elm) const {cout << elm << ' ' ;}
};

int main()
{
    int size = 5;
    vector<int> v(size);
    PrintElements print_it;
    for_each(v.begin(), v.end(), print_it);
    for_each(v.begin(), v.end(), setValue<10>);
    for_each(v.begin(), v.end(), print_it);
    return 0;
}
```

Output:

```
0 0 0 0 0 10 10 10 10 10
```

More On Template Parameters

The name used for a template parameter has not intrinsic meaning. In the previous example, the **min()**'s template name **T** could be anything:

```
template <class DontCare>
const DontCarer& min(const DontCare& a, const DontCare& b) { return a < b ? a : b; }
```

But we need to know whether the parameter is a **type** parameter or a **nontype** parameter. If it is a type parameter, then we know that the parameter represents an as yet unknown **type**. If it is a nontype parameter, we know it is an yet unknown **value**.

Scope of Template Parameter

```
typedef float T;
template <class T>
T& min(const T& a, const T& b)
{
    T tmp = a < b ? a : b;
    return tmp;
}
```

The global typedef that defines **T** as **float** is hidden by the type parameter named **T**. So, **tmp** is not a **float**, it still has the type of the template parameter **T**, not that of the global typedef. In other words, the type of **tmp** is whatever type gets bound to the template parameter **T**.

Template Parameter Name

A name used as a template parameter may not be reused within the template:

```
template <class T>
T& min(const T& a, const T& b)
{
    typedef float T; // error: redeclaration of template parameter T
    T tmp = a < b ? a : b;
    return tmp;
}
```

So, the name of a template parameter can be used only once within the same template parameter list:

```
// error: illegal reuse of template parameter name T
template <class T, classT>
T& min(const T& a, const T& b)
```

In a function template parameter list, the keywords **typename** and **class** can be used interchangeably. But when want to use types inside a function template, we must tell the compiler that the name we are using refers to a **type**. We must be explicit because the compiler cannot tell when a name defined by a type parameter is a **type** or a **value**:

```
template <typename T>
void foo(const T& vector)
{
    T::const_iterator* it;
}
```

This appears that we're declaring **it** as a pointer to a **T::const_iterator**. It's because we know that **T::const_iterator** is a **type**. But what if **T::const_iterator** isn't a **type**?

If **T::const_iterator** is a **type**, then the line is a **declaration**. But if **T::const_iterator** is an **object**, then it's a **multiplication**. We know that **const_iterator** must be a member of the type bound to **T**, but we do not know whether **const_iterator** is the name of a **type** or a **data** member. Until **T** is known,

there's no way to know whether **T::const_iterator** is a type or isn't. When the template foo is parsed, **T** isn't known. The compiler, by default, assumes that it's a data member not **type**. In other words, **T::const_iterator** is assumed to **not** be a **type**.

So, if we want the compiler to treat **const_iterator** as a type, then we must explicitly tell the compiler to do so:

```
template <typename T>
void foo(const T& vector)
{
    typename T::const_iterator* it;
}
```

By writing **typename T::const_iterator**, we indicate that member **const_iterator** of the type bound to **T** is the name of a type.

Let's take a look at similar example:

```
template
class A {
    typename T::MyType * ptr;
    ...
}
```

The **typename** is used to clarify that **MyType** is a type of class T. So, **ptr** is a pointer to the type **T::MyType**. Without **typename**, **MyType** would be considered a static member. So,

```
T::MyType * ptr
```

would be a multiplication of value **MyType** of type T with **ptr**.

Scott Meyers called a variable like **T::const_iterator** a **nested dependent type name**.

So, the general rule is: anytime we refer to a nested dependent type name in a template, we must immediately precede it by the word **typename** because in C++, any identifier of a template is considered to be a **value**, except if it is qualified by **typename**.

Check another case when we need **typename** - Generic Linked List (linkedlist.php#linkedlistexample10). Without the specification, we'll probably get the following error from Visual Studio 2010:

```
dependent name is not a type prefix with 'typename' to indicate a type
```

Let's look at another example of **typename**:

```
template<typename T>
void foo(T it)
{
    typename iterator_traits<T>::value_type tmp(*it);
}
```

It declares a local variable, `tmp`, which is the same type as what **T** objects point to, and it initializes **tmp** with the object that **T** points to. So, if **T** is **list<string>iterator**, **tmp** is of type **string**.

Let's talk about the **iterator_traits**. When we write a generic code, we may need the type of elements to which the iterator refers. So, C++ STL provides a special template structure to define the **iterator_traits**. This structure has information regarding an iterator. It is used as a common interface for all the type definitions an iterator should have, such as the category, the type of the elements, and so on:

```
template <class T>
struct iterator_traits {
    typedef typename T::value_type      value_type;
    typedef typename T::difference_type difference_type
    typedef typename T::iterator_category iterator_category
    typedef typename T::pointer         pointer;
    typedef typename T::reference        reference;
};
```

Here, **T** stands for the type of the iterator. So, the following expression yields the **value type** of iterator type **T**:

```
typename std::iterator_traits<T>::value_type
```

iterator_traits is so named because it describes argument's property/traits. The most important feature of traits templates is that they provide us a way to associate information with a type **non-intrusively**. In other words, if we have a new package of code which gives us some iterator-like types called **iter_like** that refers to a **double**, we can assign it a **value_type** without any disturbance to our existing code. All we have to do is to add an explicit specialization of **iterator_traits**, and **foo()** will see the type **double** when it asks about the **value_type** of new package's iterator:

```
namespace std
{
    template<>
    struct iterator_traits<NewPackage::iter_like>
    {
        typedef double value_type;
        ...
    };
}
```

This non-intrusive aspect of traits is exactly what makes **iterator_traits** work for pointers: the standard library contains the following **partial** specialization of **iterator_traits**:

```
template <class T>
struct iterator_traits<T*>
{
    typedef T                value_type;
    typedef ptrdiff_t        difference_type;
    typedef random_access_iterator_tag iterator_catogory;
    typedef T*               pointer;
    typedef T&               reference;
}
```

So, for any type **T***, it is defined that it has the random access iterator category, and thanks to the indirection through **iterator_traits**, generic functions can now access an iterator's associated types uniformly, whether or not it is a pointer.

Here is the summary:

1. When declaring template parameters, **class** and **typename** are interchangeable.
2. Use **typename** to identify nested dependent type names.
3. If there is any doubt as to whether **typename** is necessary to indicate that a name is a type, just specify it. There is not harm in specifying **typename** before a type.

Instantiation

Template is not itself a class or a function, it is a **blueprint**. The compiler uses the template to generate type-specific versions. The process of generating a type-specific instance of a template is known as **instantiation**. A template is instantiated when we use it: A class template is instantiated when we refer to an actual template class type, and a function template is instantiated when we call it or use it to initialize or assign to a pointer to function.

When we write

```
vector<double> vd;
```

the compiler automatically creates a class named **vector<double>**. In effect, the compiler creates the **vector<double>** class by rewriting the **vector** template, replacing every occurrence of the template parameter **T** by the type **double**.


```
template<class T>
class vector
{
    int sz;
    T* element;
public:
    vector():size(0), element(0) {}
    ~vector() { delete[] element; }
    vector &operator=(const vector&);
    const T& operator[](int n){ return element[n]; }
    void push_back(const T&d);
    ...
};
```

Template Specialization

It is not always possible to write a single template that is for every possible template argument with which the template might be instantiated. Because general definition might not compile or might do the wrong thing, we may be able to take advantage of some specific knowledge about a type to write a more efficient function than the one that is instantiated from the template.

If we want to implement different template when a specific type is passed as template parameter, we can declare a **specialization** of that template.

For example, let's suppose that we have a very simple class called **DoublePlay** that can store one element of any type and that it has just one member function called **doubleIt**, which doubles its value. But we find that when it stores an element of type **string** it would be more convenient to have a completely different implementation with a function member **mine**, so we decide to declare a class template specialization for that type

```

#include <iostream>
#include <string>

template <class T>
class DoublePlay
{
private:
    T mine;
public:
    DoublePlay(T& a): mine(a){}
    T doubleIt(){
        return mine*2;
    }
};

template <>
class DoublePlay <std::string>
{
private:
    std::string mine;
public:
    DoublePlay(std::string& a):mine(a){}
    std::string& doubleIt(){
        return mine.append(mine);
    }
};

int main()
{
    int i = 10;
    std::string s="ab";

    DoublePlay<int>myPlayA(i);
    DoublePlay<std::string>myPlayB(s);

    std::cout << i << " to " << myPlayA.doubleIt() << std::endl;
    std::cout << s << " to " << myPlayB.doubleIt() << std::endl;
}

```

Output is:

```

10 to 20
ab to abab

```

The syntax we used in the class template specialization:

```

template <> class mycontainer <std::string> { ... };

```

Notice that we precede the class template name with an empty **template<>** parameter list. This is to explicitly declare it as a template specialization. In other words, the **template<>** in declarations distinguish between the explicit **instantiation** and the explicit **specialization**.

Actually, to extend our understanding of templates, we need to investigate the term **instantiation** and **specialization**. Including a function template in our code does not in itself generate a function definition. It's merely a plan for generating a function definition. When the compiler uses the template

to generate a function definition for a particular type, the result is termed an **instantiation** of the template.

In our earlier example, the function call **min(x,y)** causes the compiler to generate an instantiation of **min()**, using **int** as a type.

The template is **not** a function definition, but the specific **instantiation** using **int** is a function definition. This type of instantiation is termed **implicit instantiation** because the compiler deduces the necessity for making the definition by noting that the code uses a **min()** function with **int** parameters.

But more important than this prefix, is the **<std::string>** specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (**string**). Notice the differences between the generic class template and the specialization:

```
template <class T> class DoublePlay { ... };  
template <> class DoublePlay <string> { ... };
```

The first line is the generic template, and the second one is the specialization.

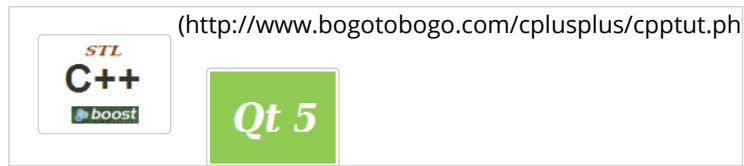
When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no **inheritance** of members from the generic template to the specialization.

Here is another example of using template specialization used as a part of template metaprogramming (boost.php#boostmpl).

```
#include <iostream>  
  
template <int n>  
struct Factorial  
{  
    enum { value = n * Factorial<n - 1>::value };  
};  
  
template <>  
struct Factorial<0>  
{  
    enum { value = 1 };  
};  
  
int main()  
{  
    std::cout << "Factorial<5>::value = " << Factorial<5>::value << std::endl;  
}
```

Bogotobogo's contents

To see more items, click left or right arrow.

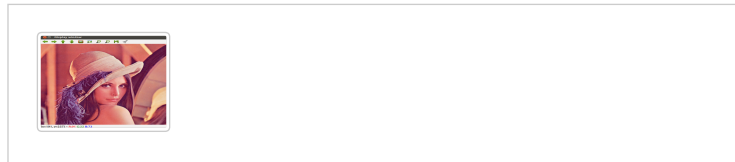


I hope this site is informative and helpful. (/about_us.php)

Bogotobogo Image / Video Processing

Computer Vision & Machine Learning

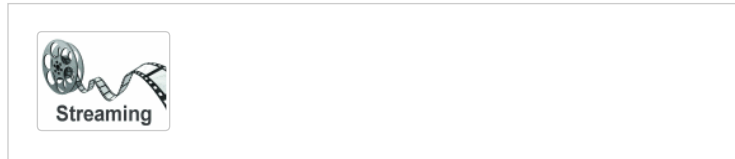
with OpenCV, MATLAB, FFmpeg, and scikit-learn.



I hope this site is informative and helpful. (/about_us.php)

Bogotobogo's Video Streaming Technology

with FFmpeg, HLS, MPEG-DASH, H.265 (HEVC)



I hope this site is informative and helpful. (/about_us.php)

CONTACT

BogoToBogo
contactus@bogotobogo.com (mailto:#)

FOLLOW BOGOTOBOGO

f (<https://www.facebook.com/KHongSanFrancisco>) **🐦**
(<https://twitter.com/KHongTwit>) **g⁺**
(<https://plus.google.com/u/0/+KHongSanFrancisco/posts>)

ABOUT US (/ABOUT_US.PHP)

contactus@bogotobogo.com (mailto:#)

Golden Gate Ave, San Francisco, CA 94115

Golden Gate Ave, San Francisco, CA 94115

Copyright © 2016, bogotobogo
Design: Web Master (<http://www.bogotobogo.com>)